Templates

Thiago Rosso Adams

http://www.thradams.com
thiago.adams@gmail.com

Introdução

Apesar de toda biblioteca padrão usar *templates*, muitas pessoas ainda têm uma visão muito superficial sobre este assunto, deixando de tirar proveito desta maravilhosa característica do C++.
Para muitos programores que vêm do C, a primeira vista os templates podem parecer substituíveis por macros. Na verdade os templates são muito mais poderosos do que macros e espero, através deste artigo, mostrar as possibilidades que se ampliam com a programação usando templates.
Este artigo apresenta a nomeclatura básica de templates e alguns conceitos como especialização completa, especialização parcial, polices/traits, embedded type information e a técnica chamada Curiously Recurring Template Pattern.

Motivação

Para iniciar vou apresentar uma pequena função e como ela seria implementada utilizando templates.
A função esccolhida é chamada de *swap* cujo trabalho é fazer a troca de conteúdo entre duas variáveis.

```
void swap(int & a, int & b)
{
	int temp(a);
	a = b;
	b = temp;
}
```

A função foi implementada para inteiros, mas desejo a mesma função outros tipos. Todas as implementações têm o mesmo código em comum com exceção do tipo do dado. Então criando uma função para cada tipo claramente estarei gerando código duplicado.
Este é um dos problemas em que os templates podem ser aplicados.
Para isso nós criamos uma ***função template*** swap que possui um ***parâmetro template*** T.

```
template<class T>
void swap(T & a, T & b)
{
	T temp(a);
	a = b;
	b = temp;
}

…
int i1 = 1;
int i2 = 2;
swap(a, b); // ok

double d1 = 1.0;
double d2 = 2.0;
swap(d1, d2); // ok
..
```

As funções template podem são ***instanciadas*** em algum ponto do fonte para um tipo T específico. Quando o tipo pode ser deduzido, por exemplo pelo tipo de argumento, a syntaxe da função é

exatamente igual a qualquer outra. Caso o tipo não possa ser deduzido ou é ambiguo podemos explicitamete informar o tipo através da sintaxe:
swap<type>(a1,a2);

De forma similar os templates podem ser usados para declaração de classes. Por exemplo:

```
template <class T>
class Array
{
        T * m_pointer;
        …
        const T & at(unsigned int index) const
        {
                return m_pointer[index];
        }
        …
};
```

Neste exemplo a classe array precisa ser instanciada para o tipo T. Ela é chamada de uma ***classe template***.

```
Array<int> array;
```

Exemplos como este são encontrados nos containers da STL. A vantagem é clara, você não precisa trabalhar de forma polimórfica com os tipos, pois o que você deseja realmente é um array de inteiros.

Especialização

Uma versão do template para um tipo em especial é chamada de ***especialização***. Existem dois tipos de especializações a ***especialização completa ou explícita*** e a ***especialização parcial***. Para toda especialização primeiramente é preciso um ***template primário*** com o caso geral.
Podemos pegar a função template swap definida anteriormente para nossa template primário e gerar uma especialização completar para um tipo específico.
Por exemplo, uma especialização para o tipo int seria:

```
template<>
void swap(int & a, int & b)
{
        cout << "swap< int>() especialization\n";
        int temp( a);
        a = b;
        b = temp;
}
```

A especializaç#ao para int será instânciada toda vez que o argumento for do tipo inteiro. Para os outros tipo o template primário será utilizado.
A ***especialização completa*** é usada para definir uma classe template para um número exato de argumentos.

Especialização Parcial

A ***especialização parcial*** é uma especilização aonde você necessita mais parâmetros para definir a sua instância do template. Neste caso a lista de parâmetros não é vazia.
É possível criar uma especialização parcial para o caso de ponteiros por exemplo ou qualquer outro caso. Voltando ao exemplo da função swap, ela será usada aqui para a especialização para o tipo Array.

Apenas para deixar clara a diferença, uma especialização completa para a classe Array poderia ser escrita assim:

```
template< >
void swap<Array<int> & a, Array<int> & b>
{
  […]
}
```

Neste caso a especialização completa só funcionaria para o caso de um Array<int>.
Para resolver a especialização para qualquer Array de tipo T, pode ser usada a especialização parcial .
Ficando assim:

```
template< class T >
void swap( Array<T> & a, Array<T> & b )
{
        a.swap(b);
}
```

aonde Array::swap poderia ser definida assim:

```
void Array< T >::swap( Array<T> & b )
{
        T * tmp( m_pointer);
         m_pointer = b.m_pPointer;
         b.m_pointer = tmp;
}
```

Agora a função swap pode ser usada em qualquer Array e a especialização tornou a função muito mais eficiente e segura para o caso da classe Array, aonde uma simples cópia do ponteiro resolve o problema. Caso o template primário fosse usado seria muito mais dispendioso criar uma cópia da classe para fazer o swap. Este fator foi o que tornou interessante esta especialização partial.

Embedded Type Information

Embedded Type Information é a capacidade de um tipo armazenar informações relevantes a si próprio. Esta capacidade está intimamente ligada aos templates pois podem fornecer a eles a informação de que necessitam para sua instância. Esta informação é colocada em forma de typedefs.

Os containers da STL são exemplos de classes com embedded type information.
Pegando o vector como exemplo, encontramos nele os seguintes typedefs:

allocator_type , const_iterator, const_pointer, const_reference, const_reverse_iterator, difference_type, iterator, pointer, reference, reverse_iterator, size_type, value_type.

Estes typedefs informam o tipo de iterator, o tipo do allocator, o tipo de dado usado etc. Eles são utilizados em algorítmos genéricos. Podemos perguntar para o container “T”:
Qual é o seu iterator? Qual o tipo de allocator você utiliza?

Por exemplo:

```
template<class T> void printAll(T & container)
{
        T::iterator it = container.begin();
        for ( ;it != container.end(); ++it)
        {
                cout << *it;
        }
```

```
}
```

Percebam que caso a informação do tipo do iterator não estivesse declarada no container eu precisaria de um parâmentro extra na minha função template.

Traits

Algumas vezes a informação contida em um tipo não é suficiente para a implementação de um template. Outras vezes, o tipo não possui e não pode conter nenhuma informação extra, como é o caso dos tipos básicos int, double, etc. Os Traits podem ser usados nestes casos.
Traits, é uma pequena classe que traz informações sobre um tipo e/ou informações de como lidar com ele.
Podem ser usados em outras classes ou em algorítmos.

Exemplo:
Vamos supor que eu queria percorrer um container de ponteiros e deletar todos os items:

```
template<class T>
void DeleteAllItems(T & container)
{
        T::iterator it = container.begin();
        for ( ;it != container.end(); ++it)
        {
                delete *it;
        }
        container.clear();
}
```

Esta função template funciona perfeitamente para um ponteiro convencional. No entanto eu posso ter o mesmo algorítmo para deletar uma lista de objetos COM. (No COM é usado um método Release da interface IUnknown ao invés do operator delete)
Neste caso preciso tratar os diferentes tipos de ponteiros. Esta informação pode ser colocada externamente com a utilização de Traits.
Um Traits que contenha informação de como deletar o ponteiro pode ser definido assim:

```
// caso genérico
template<class T> void DeleteItemTraits(T p)
{
        delete p;
}

// especializando para o caso do IUnknown
template<> void DeleteItemTraits(IUnknown *p)
{
        p->Release(); // caso de um ponteiro COM
}
```

E o algorítmo genérico pode ser escrito:

```
template<class T> void DeleteAllItems(T & container)
{
        T::iterator it = container.begin();
        for ( ;it != container.end(); ++it)
        {
                DeleteItemTraits<T::value_type>(*it);
        }
        container.clear();
```

```
}
```

O termo Police também é usado para Traits, para dar a noção de ações sobre o tipo, e não apenas informações. Poderia chamar minha função de DeletePolice por exemplo.
A STL possui vários exemplos de Traits. Um destes é a implementação da std::numeric_limits.
A numeric_limits possui a função max() que retorna o máximo valor que pode ser contido no tipo.

Exemplo:

```
cout
        << "The maximum value for type int is:  "
         << numeric_limits<int>::max( ) //*
        << endl;
```

Neste caso não era possivel colocar a informação do valor de max dentro do tipo int. Por isso o conceito de traits foi usado.
Traits também são usados na STL em classes como a basic_string que precisa de um Traits para tratar o caso do tipo de caractere usado (char ou wchar_t).

Curiously Recurring Template Pattern

O nome “Curiously Recurring Template Pattern” ou CRTP é empregado para um técnica que faz com que a classe base derive de outra classe cujo parâmetro template é a própria classe derivada.

Ou seja:

```
class derived : public base<derived>
{
        …
}
```

Em poucas palavras, esta técnica permite que a classe base acesse a classe derivada através de um cast de seu ponteiro.
A maneira mais comum da classe base acessar a classe derivada é através de funções virtuais. Com a CRTP existe uma alternativa para conseguir um comportamento semelhante sem a necessidade do overhead da vtable.
Para exemplificar, vou iniciar com uma solução baseada em funções virtuais:

```
class base
{
        virtual void do_on_change()
        {
                /*default code*/
        }

public:
        void on_change() { do_on_change(); }
};

class derived : public base
{
        void do_on_change()
        {
                /*new code for do_on_change*/
        }
};
```

```
int main()
{
        derived d;
        base & b = d;
        b.on_change();
}
```

O código acima mostra o comportamento básico das funções virtuais e não requer grandes explicações. Pelo exemplo, a função do_on_change implementada na classe derivada é chamada pela classe base.

Prosseguindo, agora um exemplo equivalente utilizando a técnica CRTP.

```
template<class T>
class base
{
public:
        void on_change()
        {
                T * p = static_cast<T*> this;
                p->do_on_change();
        }
};

class derived : public base<derived>
{
public:
        void do_on_change()
        {
                ...
        }
};

int main()
{
        derived d;
        base<derived> & b = d;
        b.do_change();
}
```

A chamada da função do_on_change() da classe derivada é feita diretamente pelo ponteiro “this” da classe base convertido para classe derivada.
Apesar de parecer um pouco estranho a validade do cast é garantida justamente pela herança que foi criada.
Uma das vantagens do uso desta técnica sobre a solução com funções virtuais é justamente não precisar do overhead da vtable e das chamadas de funções virtuais. Ela tem uma caracterisca estática de montagem dos tipos, enquanto as funções virtuais tem uma característica dinâmica. Outra vantagem é que você pode acessar variáveis e enumerações da classe derivada e não apenas funções como é o caso da solução com funções virtuais.
Por exemplo:

```
template<class T>
class Algorithm
{
        enum
```

```
        {
                constValue1 = 1
        };

public:
        void DoSomething()
        {
                const int value1 = static_cast<T&>(*this).constValue1;
                 cout << value1;
                //…
   }
};

class MyAlgorithm : public Algorithm<MyAlgorithm>
{
public:
         enum
        {
                constValue1 = 2
        };
};


int main()
{
        MyAlgorithm a;
        a.DoSomething();
        return 0;
}
```

O “curious” do nome vem do fato da classe precisar dela mesma para existir. Isto só é possível pois os templates só existem após instanciados. (Não confunda instância de template com instância de objeto) A técnica de CRTP não substitue as funções virtuais de forma alguma, e não é equivalente em todas as situações. O uso mais geral da técnica depende da criatividade de cada um, mas é alternativa leve e interessante para muitos problemas. Vários exemplos desta técnica podem ser encontrados nas bibliotecas ATL e WTL.

Referências

Bjarne Stroustrup, The C++ Programming Language
Stephen C. Dewhurst, C++ Common Knowledge